Chapter 10
Modularizing Ontologies

Mathieu d'Aquin

Abstract As large monolithic ontologies are difficult to handle and maintain, the activity of modularizing an ontology consists in identifying components (modules) of this ontology that can be considered separately while they are interlinked with other modules. The end benefit of modularizing an ontology can be, depending on the particular application or scenario, (a) to improve performance by enabling the distribution or targeted processing, (b) to facilitate the development and maintenance of the ontology by dividing it in loosely coupled, self-contained components or (c) to facilitate the reuse of parts of the ontology. In this chapter, we present a brief introduction to the field of ontology modularization. We detail the approach taken as a guideline to modularize existing ontologies and the tools available in order to carry out this activity.

10.1 Motivation

In complex domains such as medicine, ontologies can contain thousands of concepts. Examples of such large ontologies are the NCI (National Cancer Institute) Thesaurus[1] with about 27,500 and the Gene Ontology[2] with about 22,000 concepts. However, problems with large monolithical ontologies in terms of reusability, scalability, and maintenance have led to an increasing interest in techniques for dividing ontologies into sets of cohesive, self-contained modules; for extracting modules from ontologies relevant to a sub-domain or a task; as well as for

[1] http://ncit.nci.nih.gov/

[2] http://www.geneontology.org/

M. d'Aquin (✉)
Knowledge Media Institute (KMi), The Open University, Walton Hall, Milton Keynes, MK7 6AA, UK
e-mail: m.daquin@open.ac.uk

combining and manipulating ontology modules. We observe however that there is no universal way to modularize an ontology and that the choice of a particular technique or approach should be guided by the requirements of the application or scenario relying on modularization.

In particular, ontologies that contain thousands of concepts cannot be created and maintained by a single person. The broad coverage of such large ontologies normally requires a team of experts. In many cases, these experts will be located in different organizations and will work on the same ontology in parallel. In other situations, large ontologies are mostly created to provide a standard model of a domain to be used by developers of individual solutions within that domain. While existing large ontologies often cover a complete domain, the providers of individual solutions are often only interested in a specific part of the overall domain.

Also, the nature of ontologies as reference models for a domain requires a high degree of quality of the respective model. Representing a consensus model, it is also important to have proposed models validated by different experts. In the case of large ontologies, it is often difficult, if not impossible, to understand the model as a whole.

On a technical level, very large ontologies cause serious scalability problems. The complexity of reasoning about ontologies is well known to be critical even for smaller ontologies. In the presence of ontologies like the NCI Thesaurus, not only reasoning engines but also modelling and visualization tools reach their limits. Currently, there is no modelling tool that can provide convenient modelling support for ontologies of the size of the NCI Thesaurus.

All these problems are a result of the fact that a large ontology is treated as a single monolithic model. Most problems would disappear if the overall model consists of a set of coherent modules about a certain sub-topic that can be used independently of the other modules while still containing information about its relation to these other modules.

In the next sections, we describe a general guideline to the modularization of ontologies and tools that can be used to support this activity. We identify three approaches which can be involved in realizing the modularization of an ontology: ontology partitioning, ontology module extraction and ontology module composition.

10.2 Ontology Modularization

We consider an ontology O as a set of axioms (sub-class, equivalence, instantiation, etc.) and the signature Sig(O) of an ontology O as the set of entity names occurring in the axioms of O, that is, its vocabulary. As described in the NeOn Glossary (Suárez-Figueroa 2010), ontology modularization refers to the activity of identifying one or more modules in an ontology. A module is considered to be a significant and self-contained sub-part of an ontology. Therefore, a module $M_i(O)$ of an ontology O is also a set of axioms (an ontology), with the minimal constraint that $Sig(M_i(O)) \subseteq Sig(O)$. Note that, while it may often be desirable, it is not always the case that $M_i(O) \subseteq O$.

10.2.1 Ontology Partitioning

The activity of partitioning an ontology consists of splitting up the set of axioms into a set of modules $\{M_1, \cdots, Mk\}$ such that each M_i is an ontology, and the union of all modules is semantically equivalent to the original ontology O (see Fig. 10.1). Note that some approaches being labelled as partitioning methods do not actually create partitions, as the resulting modules may overlap. There are several methods for ontology partitioning that have been developed for different purposes.

The method of MacCartney et al. (2003) aims at improving the efficiency of inference algorithms by localizing reasoning. For this purpose, this technique minimizes the shared language (i.e. the intersection of the signatures) of pairs of modules. A message passing algorithm for reasoning over the distributed ontology is proposed for implementing resolution-based inference in the separate modules. Completeness and correctness of some resolution strategies is preserved, and others trade completeness for efficiency.

The method of Cuenca Grau et al. (2005) partitions an ontology into a set of modules connected by ε-connections. This approach aims at preserving the completeness of local reasoning within all created modules. This requirement is supposed to make the approach suitable for supporting selective use and reuse since every module can be exploited independently of the others.

A tool that produces sparsely connected modules of reduced size was presented in Stuckenschmidt and Klein (2004). The goal of this method is to support maintenance and use of very large ontologies by providing the possibility to individually inspect smaller parts of the ontology. The algorithm operates with a number of parameters that can be used to tune the result to the requirements of a given application.

Later in this chapter, we describe a method for ontology partitioning based on enforcing good properties in the dependency graph between the resulting modules.

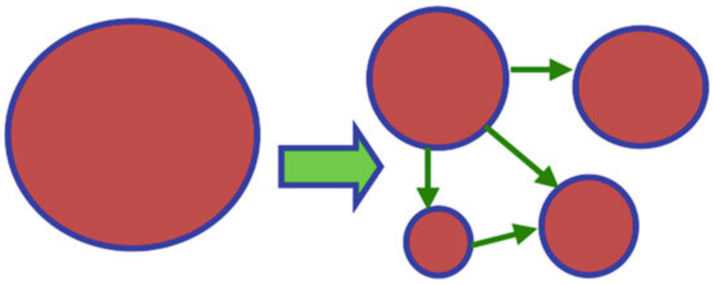


Fig. 10.1 Ontology partitioning

10.2.2 Ontology Module Extraction

Ontology module extraction consists in reducing an ontology to the sub-part, the module, that covers a particular sub-vocabulary. This activity has been called segmentation in Seidenberg and Rector (2006) and traversal view extraction in Noy and Musen (2004). More precisely, given an ontology O and a set SV $\subseteq$ Sig(O) of terms from the ontology, a module extraction mechanism returns a module M_{SV}, supposed to be the relevant part of O that covers the sub-vocabulary SV (Sig(M_{SV}) $\supseteq$ SV, see Fig. 10.2). Techniques for module extraction often rely on the so-called traversal method: starting from the elements of the input sub-vocabulary, relations in the ontology are recursively 'traversed' to gather relevant (i.e. related) elements to be included in the module.

Such a technique has been integrated in the PROMPT tool (Noy and Musen 2004), to be used in the Protégé environment. This method recursively follows the properties around a selected class of the ontology until a given distance is reached. The user can exclude certain properties in order to adapt the result to the needs of the application.

The mechanism presented in Seidenberg and Rector (2006) starts from a set of classes of the input ontology and extracts related elements on the basis of class subsumption and OWL restrictions. Some optional filters can also be activated to reduce the size of the resulting module. This technique has been implemented to be used in the Galen project and relies on the Galen upper ontology.

In Stuckenschmidt (2006), the author defines a viewpoint as being a sub-part of an ontology that only contains the knowledge concerning a given sub-vocabulary (a set of concept and property names). The computation of a viewpoint is based on the definition of a viewpoint-dependent subsumption relation.

Inspired from the previously described techniques, d'Aquin et al. (2006) define an approach for the purpose of the dynamic selection of relevant modules from online ontologies. The input sub-vocabulary can contain classes, properties or individuals. The mechanism is fully automatized, is designed to work with different kinds of ontologies (from simple taxonomies to rich and complex OWL ontologies), and relies on inferences during the modularization process.

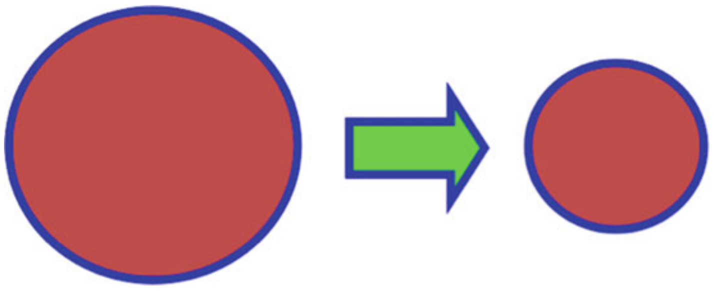


Fig. 10.2 Ontology module extraction

Finally, the technique described in Doran et al. (2007) is focussed on ontology module extraction for aiding an ontology engineer in reusing an ontology module. It takes a single class as input and extracts a module about this class. The approach it relies on is that, in most cases, elements that (directly or indirectly) make reference to the initial class should be included.

One important issue related to ontology module extraction is that different scenarios and applications require different ways to modularize ontologies (d'Aquin et al. 2007b). To facilitate the selection, combination and adaptation of the various existing module extraction techniques, d'Aquin et al. (2007a) describe a parametric approach for module extraction. The principle is to describe module extraction techniques under a common framework that can be parameterized according to the modularization technique that is most suited for the application. This framework relies on a graph transformation engine. Ontologies to be modularized are represented as graphs, and modularization techniques re-formulated as graph transformation rules. In this way, existing modularization technique can be implemented in the same tool, making it easier to compare, adapt and combine them, and new modularization techniques can easily be implemented in the form of modularization rules. The paper (d'Aquin et al. 2007a) described the reformulation of several existing techniques for modularization, but an operational implementation of the tool has not been made available.

Very similar ideas to the one described in d'Aquin et al. (2007a) are at the basis of another approach for parametric modularization (Doran et al. 2008) which, instead of a graph transformation framework, employs a mechanism that recursively execute SPARQL queries over the ontology to build a sub-set of it. The parameters of this framework are the sets of SPARQL queries that represent modularization techniques. In the same line of ideas, we describe later in this chapter a tool that relies on a set of specific extraction operators that can be combined to extract modules from ontologies in a way that is customized to the application at hand.

10.2.3 Ontology Module Composition

In Wiederhold (1994), Wiederhold defines a simple ontology algebra, with the main purpose of facilitating ontology-based software composition. He defines a set of operators applying set-related operations on the entities described in the input ontologies and relying on equality mappings (=) between these entities. More precisely, the three operators are defined as shown in Table 10.1.

In the same line of ideas, but in a more formalized and sophisticated way, Melnik et al. (2004) describe a set of operators for model management, as defined in the Rondo platform (Melnik et al. 2003). The goal of model management is to facilitate and automatize the development of metadata-intensive applications by relying on the abstract and generic notion of model of the data, as well as on the idea of mappings between these models. An essential part of a platform for model

Table 10.1 Set of operators

Intersection(O_1, O2) → O	Creates an ontology O containing the common (mapped) entities in O_1 and O_2
Union(O_1, O_2) → O	Creates an ontology O containing the entities of O_1 and O_2 and merging the common ones
Difference(O_1, O_2) → O	Creates an ontology O containing only the entities in O_1 that are not mapped to entities in O_2

management is a set of operators to manipulate and combine these models and mappings. Melnik et al. (2004) focus on formalizing a core set of operators: Match, Compose, Merge, Extract, Difference and Confluence. Match is particular in this set. It takes two models as an input and returns a mapping between these models. It inherently does not have a formal semantics as it depends on the technique used for matching, as well as on the concrete formalism used to describe the models and mappings. Merge intuitively corresponds to the Union operator in Wiederhold (1994): it takes two models and a mapping and creates a new model that contains the information from both input models, relying on the input mapping. It also creates two mappings from the created model to the two original ones. Extract creates the sub-model of a model that is involved in a mapping, and Difference, the sub-model that is not involved in a mapping. Finally, compose and confluence are mapping manipulation operators, creating mappings by merging or composing other mappings.

Kaushik et al. (2006) define operators for combining ontologies created by different members of a community and written in RDF. This paper first provides a formalization of RDF to describe set-related operators such as Intersection, Union and Difference. It also adds other kinds of operators, such as the quotient of two ontologies O_1 and O_2 (collapsing O_2 into one entity and pointing all the properties of O_1 to entities of O_2 to this particular entity) and the product of two ontologies (inversely, extending the properties from O_1 to O_2 to all the entities of O_2). It is worth mentioning that such operators can be related to the ones of relational algebras used in relational database systems.

Note finally that the OWL tools[3] that are part of the KAON2[4] framework include operators such as Difference, Merge and Filter, working at the level of ontology axioms. For example, merge creates an ontology as the union of the axioms contained in the two input ontologies. The NeOn Toolkit plugin for ontology module composition presented in Sect. 10.5.3 relies on similar simple operators and is integrated with the other tools for module extraction and ontology partitioning.

[3] http://km.aifb.kit.edu/projects/owltools/

[4] http://kaon2.semanticweb.org/

10.3 A General Approach to Modularizing Ontologies

As we mentioned in Sect. 10.1, the goal of ontology modularization is to obtain a module or a set of modules from an ontology, which fit the requirements of a particular application or a particular scenario. Especially due to the large number of different techniques that can be used and combined to achieve these goals, there is a need for methodological guidelines to help ontology developers in selecting and applying the appropriate techniques for modularization, depending on the goal of modularization.

Note that, as opposed to a single, monolithic ontology, an ontology network is essentially a modular ontology, made of components (the individual ontologies) interacting with each other in a particular context. The approach presented here is applied on individual ontologies (possibly networked) to create either networks of ontologies or elements for networks of ontologies.

Generalizing and clarifying the description above, we specify the definition of ontology modularization, as provided by the NeOn Glossary (Suárez-Figueroa 2010), as the activity that takes as an input an ontology and that has for goal to identify a set of modules for this ontology, effectively creating a modular version of it, for the purpose of supporting maintenance and reuse (see Fig. 10.3). Modularization offers a way to cut down potentially large ontologies into smaller, more manageable modules. It is generally realized by the ontology engineer or the ontology engineering team, preferably with the help of domain experts.

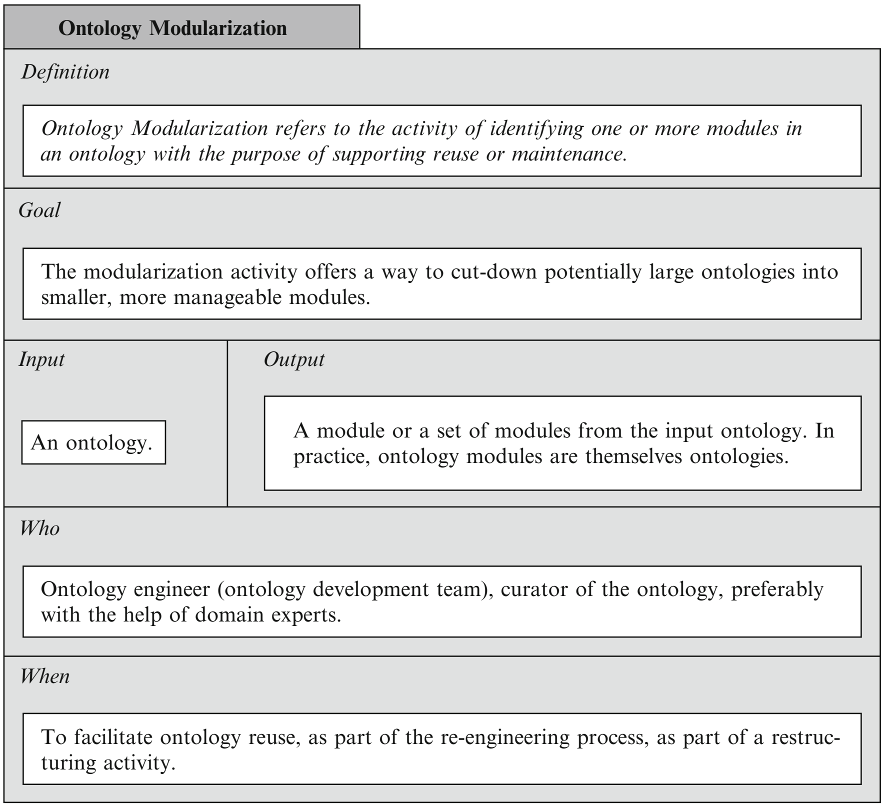
Figure 10.4 shows the workflow and the tasks for carrying out the ontology modularization activity. As can be seen in this figure, we see this activity as an iterative process, potentially combining different methods and techniques for module extraction and partitioning, and combining their results through the use of module composition operators.

Task 1. Identifying the Purpose of Modularization

As discussed earlier, the modularization of an ontology strongly depends on the application relying on the modularization and the context in which the ontology is developed. It is therefore crucial to start by identifying the reasons for modularizing the ontologies and the expected benefits, to guide the rest of the process.

Commonly considered benefits (and thus drivers) of ontology modularization are:

- *Improving performance* by enabling the distribution of reasoning or by exploiting only the relevant modules of a large ontology (see Suntisrivaraporn et al. (2008) for an example in inference justification)
- *Facilitating the development and maintenance of the ontology* by dividing it in loosely coupled, self-contained components, which can be managed separately
- *Facilitating the reuse of (parts of) the ontology* by extracting modules of the ontology that have a specific application or purpose for being reused
- *Customizing ontologies* by application developers to flexibly extract and combine modules relevant to a particular application or to provide different modules



Ontology Modularization

Definition

Ontology Modularization refers to the activity of identifying one or more modules in an ontology with the purpose of supporting reuse or maintenance.

Goal

The modularization activity offers a way to cut-down potentially large ontologies into smaller, more manageable modules.

Input

An ontology.

Output

A module or a set of modules from the input ontology. In practice, ontology modules are themselves ontologies.

Who

Ontology engineer (ontology development team), curator of the ontology, preferably with the help of domain experts.

When

To facilitate ontology reuse, as part of the re-engineering process, as part of a restructuring activity.

Fig. 10.3 Ontology modularization filling card

to different groups of users (see Lopez et al. (2009) for an example in managing access rights in a distributed question answering system)

Identifying the purpose of modularization is essential for the next tasks, in particular to select the appropriate modularization technique and criteria to maximize the expected benefit of modularization.

Task 2. Selecting a Modularization Approach

As explained at the beginning of this chapter, there are two main approaches to obtain modules from ontologies: ontology partitioning and ontology module extraction. It is generally easy to decide which one to choose according to the modularization purpose:

- Whenever the purpose relates to the entire ontology (i.e. improving maintenance, and in some cases performance), a partitioning approach should be considered.
- Whenever the purpose relates to extracting specific parts of an ontology (e.g. to customize it or reuse it partially), module extraction should be considered.

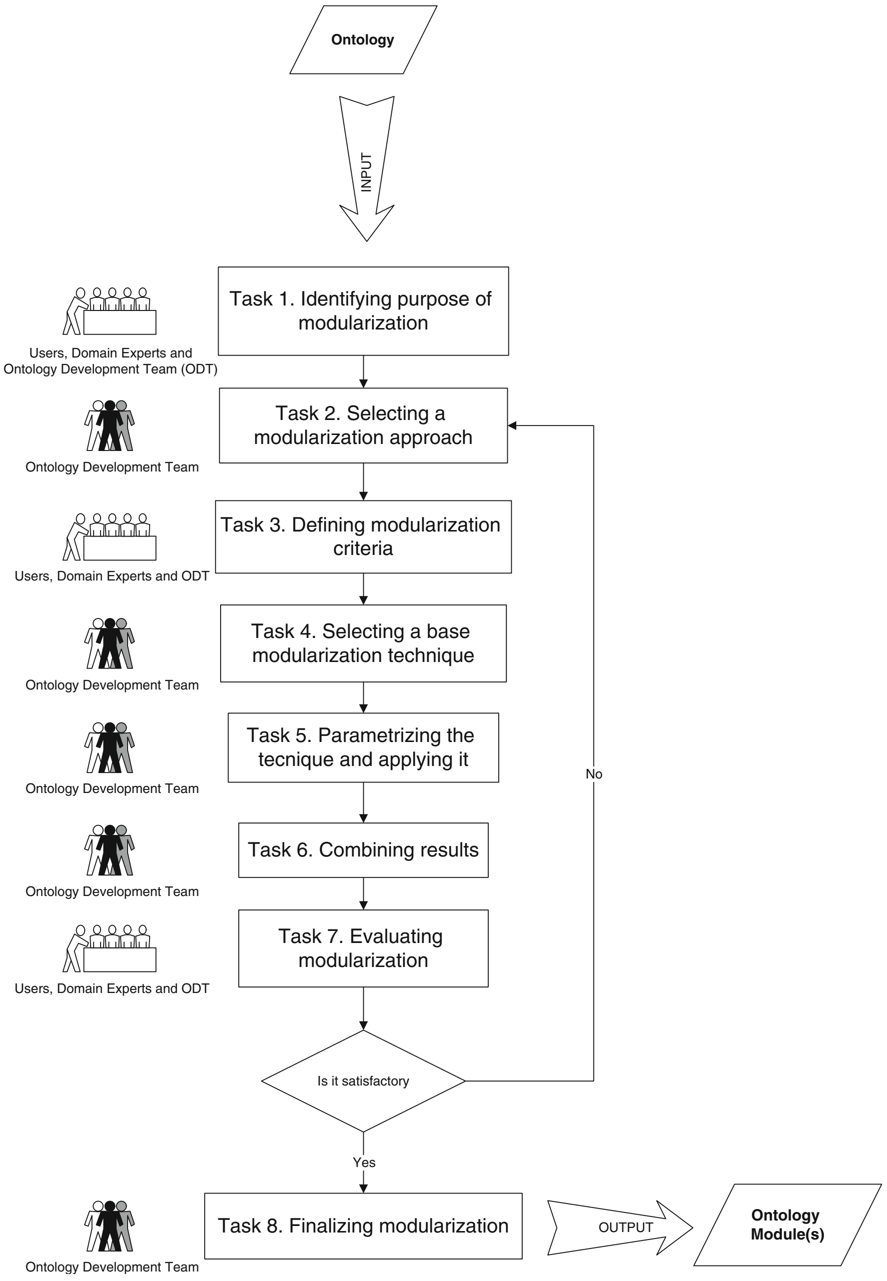

Ontology
INPUT
Task 1. Identifying purpose of modularization
Users, Domain Experts and Ontology Development Team (ODT)
Task 2. Selecting a modularization approach
Ontology Development Team
Task 3. Defining modularization criteria
Users, Domain Experts and ODT
Task 4. Selecting a base modularization technique
Ontology Development Team
Task 5. Parametrizing the tecnique and applying it
Ontology Development Team
Task 6. Combining results
Ontology Development Team
Task 7. Evaluating modularization
Users, Domain Experts and ODT
Is it satisfactory
No
Yes
Task 8. Finalizing modularization
Ontology Development Team
OUTPUT
Ontology Module(s)

Fig. 10.4 Tasks for modularizing ontologies

Of course, this needs to be considered in the context of the overall iterative process that constitutes ontology modularization. In general, when the purpose is to obtain a set of modules to cover the entire ontology, in a first iteration, partitioning should be considered. In subsequent iterations, intermediary modules might need to be further partitioned, or specific modules be extracted.

Task 3. Defining Modularization Criteria

The modularization criteria define the basic characteristics that the resulting modules should have, that is, what should go into a module. In d'Aquin et al. (2009), a set of criteria typically employed for modularization is given (e.g. logical completeness and correctness with respect to the original ontology, size, relation between modules). The criteria to emphasize should be decided, depending on the purpose of modularization (as defined in Task 1). For example, if the goal is to improve the reasoning procedure, logical criteria should be favoured. In d'Aquin et al. (2009), we showed that the great variety of techniques for modularization all implement different criteria, meaning that this task is essential for choosing the appropriate technique, or combination of techniques. Unfortunately, while work in d'Aquin et al. (2009) provides a list of common criteria, and insights on their importance in different scenarios, the choice of the right criteria to apply is highly dependent on a particular situation and has to be left to the ontology engineers to decide.

Task 4. Selecting a Base Modularization Technique

As mentioned in previous sections, there is a great variety of techniques and tools for ontology modularization. In d'Aquin et al. (2007b, 2009), we showed that these techniques implement a *different intuition* about what should be in a module, and so, there is no universal definition of what an ontology module should contain. In other words, it is necessary to select the most appropriate technique, depending on the criteria to apply. There is currently no comprehensive list of techniques that could be applied for modularization. However, authors in d'Aquin et al. (2009) provide a description of the major techniques and experiments, demonstrating how they realize some possible criteria.

Task 5. Parametrizing the Technique and Applying It

Depending on the technique that has been selected by Task 4, there may be various parameters required to obtain interesting and useful results. For example, module extraction techniques generally require identifying a sub-vocabulary of the original ontology, defining a particular area of interest. Partitioning techniques may require indications, for example, about the minimal/maximal size of a module. In such cases, the ontology engineer can only refer to guidelines and manual of the individual tool to establish the best parameters in his/her context. Most of the techniques would, in principle, be applied in the same way, taking the original ontology as input and creating modules in the form of smaller ontologies, allowing in this way to process the resulting modules iteratively, in the same way as the original ontology.

Task 6. Combining Results

As mentioned earlier, we favour an iterative process where the adequate modules are produced by refining and combining the results obtained with various parameters, techniques and approaches. Therefore, at every iteration, everytime a new (set of) module(s) is produced, it is necessary to integrate it – that is, to combine it – with the modules that were produced at previous iterations. The way to combine depends on the criteria for modularization and on the modules already produced. Two possibilities are:

- If some modules were too small or not logically complete and the current iteration produced complementary modules, then the results should be merged.
- If modules from a previous iteration were too big because the employed technique did not consider some of the criteria, and a new technique is applied that implements the missing criteria, then the common part from the results of both iteration should be considered.

Operators for combining modules should be employed here to derive new modules from the results of partitioning or extraction techniques, or from different iterations for the process. The three common operators should be applied in the following situations:

- *Intersection:* when two or more modules have been produced that are complementary in the sense that they are too broad and should be reduced in relation with each other
- *Union:* when two or more modules have been produced that are complementary in the sense that they are too narrow and should be integrated with each other
- *Difference:* when two or more modules have been produced that are complementary in the sense that one should be narrowed down so that it does not overlap with the other

Task 7. Evaluating Modularization

The evaluation of the result of the modularization (meaning the complete set of generated modules to be included in the modular ontology) is a crucial part of the iterative process. Indeed, it depends on this evaluation whether a new iteration is necessary, applying a new set of criteria and a new technique, or if the current (set of) modules are satisfactory, considering the application scenario. There are two ways in which the modularization could be evaluated:

- *By checking the criteria:* Evaluating whether the criteria defined for modularization have been realized as expected by the modularization technique is useful both for checking if the results match the requirements of the application and for establishing a new set of criteria in case another iteration is required.
- *By testing against the purpose of modularization:* If the defined criteria have all been realized, it is important to check whether or not the obtained modularization actually realizes the expected improvement compared to the original ontology. For example, if the goal was to facilitate the maintenance of the ontology, the ontology engineers and domain experts should check whether the structure of the new, modular ontology has been created in a sensible way according to this purpose. Another example could be when the goal is to better support an application; in these cases, further guidelines about how to perform an application-based ontology evaluation can be found in Chap. 9.

There can be three outcomes for this task. It can establish by evaluation that:

- *The modularization is satisfactory*, so that the created modules can be finalized and deployed (Task 8).
- *The modularization is incomplete*, so that a new iteration should be carried on, using another set of criteria and another technique to produce complementary results.
- *The modularization is improper*, so that a new iteration is required, re-considering the set of criteria and the technique to employ in order to produce modules that better match the purpose of modularization.

Note that in different iterations, only the purpose of modularization cannot change. In particular, even if the approach (extraction or partitioning) generally does not change, it is not hard to imagine scenarios in which a partitioning technique is first applied, followed by extraction procedures on the previously created modules, as showed by the example in Sect. 10.4.

Task 8. Finalizing Modularization

Once the produced modularization is judged satisfactory, an additional step can be required for it to be deployed and exploited in an application. For example, it is usually necessary to revise the identifiers of each of the modules so that they follow the conventions employed in the target application, to re-establish links between modules, or simply to deploy the resulting modules in a way that it is made accessible in the target application and the editorial workflow.

10.4 Example

We consider the scenario where a large monolithic ontology has been developed in the past, and this needs to be modularized in order to facilitate its maintenance. The purpose of the modularization has therefore been clearly identified (Task 1). In this case, it is clear that what is required is to produce a set of modules that together cover the entire ontology. Thus, in Task 2, the partitioning approach is selected. Considering that the purpose is to facilitate maintenance, the major criteria (Task 3) to take into account are:

- The sizes of the modules, which should be small enough to be easily manageable but not too small so that the ontology curator does not have to handle too many different modules for a particular management task
- The relations between modules, which should favour a well-structured organization in the dependency of the modules

Considering both criteria above, it is decided to apply the NeOn Toolkit plugin for ontology partitioning (see Sect. 10.5), which works on the dependency graph of modules and intends to provide good structures for this dependency graph (Task 4). The only parameter for this technique is the minimum size of a module (Task 5), which is chosen according to the size of the initial ontology. The resulting partition

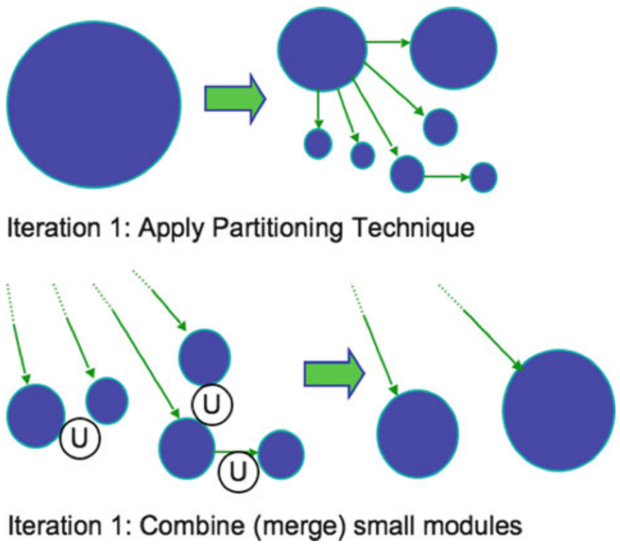

Iteration 1: Apply Partitioning Technique
U
U
U
Iteration 1: Combine (merge) small modules

Fig. 10.5 First iteration in our example modularization process

is described in Fig. 10.5. Even if there are no previous results yet, some modules produced by the partitioning technique can already be combined together (Task 6). Indeed, small modules can be judged too small and might contain information that is considered relevant for other modules. Therefore, these modules can be merged using the NeOn Toolkit plugin for module combination and employing the Union operator. This is depicted for our example in Fig. 10.5.

Now that a first result has been produced, it can be evaluated (Task 7) by the ontology development team, the domain experts and the users. In this example, there is one module that is considered too big and covering two different topics that should be separated. A second iteration is necessary.

The goal of the second iteration is to extract from one of the modules produced previously, the elements related to one particular topic. Thus, we chose to follow the extraction approach (Task 2). The criteria here are mainly that the extracted module should contain ontological elements relevant to this particular topic (Task 3). A specific ontology module extraction technique is selected for this (Task 4) and used to generate relevant modules on the basis of a set of core terms defining the topic (Task 5). The result is depicted in Fig. 10.6. Now that one module has been extracted for one of the topic covered by the original module, the one for the second topic has to be created in the combination task (Task 6). This is achieved by using the Difference operator in the module combination plugin of the NeOn Toolkit (see Fig. 10.6). In this way, the original module has then been divided into two modules, one being the complement of the other. We then obtain a new set of modules that can be evaluated, and if judged adequate, can replace the original, monolithic ontology.

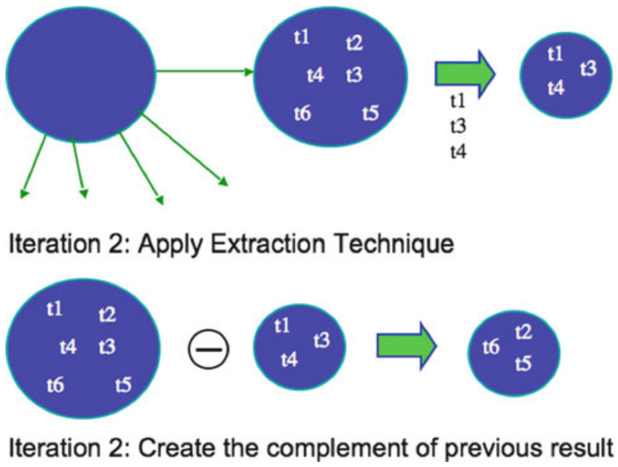

t1
t2
t4
t3
t6
t5
t1
t3
t4
t1
t3
t4
Iteration 2: Apply Extraction Technique
t1
t2
t4
t3
t6
t5
t1
t3
t4
t6
t2
t5
Iteration 2: Create the complement of previous result

Fig. 10.6 Second iteration in our example modularization process

10.5 Tool Support

The abstract example presented above provides an illustration of the overall activity of modularizing an existing ontology, using the iterative method we propose, based on different modularization approaches, and combining results from different techniques. Ideally, the tools necessary to achieve this activity of modularizing should be integrated within the same ontology engineering environment in which the ontologies are developed. Here, we present the tools integrated in the NeOn Toolkit in order to realize ontology partitioning, ontology module extraction and ontology module composition. Together with the NeOn Toolkit, these tools represent an integrated environment for creating and manipulating ontology modules.

10.5.1 Ontology Partitioning

Our method for ontology partitioning is based on basic requirements concerning the resulting modularization and its structure. We consider that the result of the partitioning process should not only be a bag of modules but should also provide the relations between them in terms of dependency. In addition, some good properties for this structure should be enforced in order to facilitate the manipulation and maintenance of the modularization.

As our approach is based on the dependency structure of modules, we need to define this relation of dependency. We consider a module M_1 to be dependent on a module M_2 if there is at least one entity in M_1 whose definition or description depends on at least one entity in M_2. The definition or the description of an entity

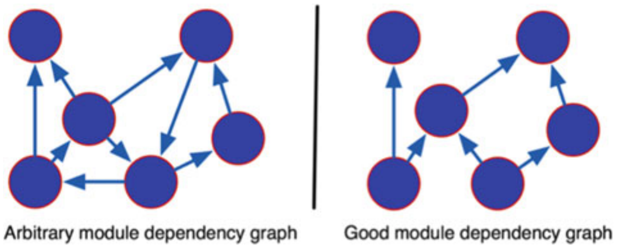

Arbitrary module dependency graph
Good module dependency graph

Fig. 10.7 Graphs illustrating dependency structures between ontology modules

A depends on an entity B whenever B participates in the axioms defining or describing A.

From this definition, we can see that if a module M_1 depends on a module M_2, it means that M_1 should import M_2. The main particularity of our approach is that we want the dependency structure of the resulting modularization to have good properties in order to be efficient in facilitating further engineering of the obtained modular ontology. In other terms, as shown in Fig. 10.7, we do not want this structure to be any arbitrary (directed) graph, but to respect two major rules:

1. *Rule 1 (no cycle)*: There should not be any cycle in the dependency graph of the resulting modularization. The rationale for this rule is that we are trying to reproduce the natural situation where modules would be reused. Creating bidirectional interdependencies between reused modules is a bad practice as it introduces additional difficulties in case of an update of one of the modules or when distributing modules (Parnas 1978).
2. *Rule 2 (no transitive dependency)*: If a module reuses another one, it should not directly or indirectly reuse a module on which the reused one is dependent. Indeed, when this situation arises, it means that the organization of modules into layers has not been enforced, so that a module is reusing other modules at different levels of the same branch of the dependency graph. Besides producing unnecessary redundancies in the dependency structure, this could also cause difficulties for the evolution and distribution of the module by creating 'concurrent propagation paths,' leading to the same module.

In addition, in order to ensure not only that the structure of the modularization respects good properties but also that individual modules are easy to manage and to handle, we add two rules on the characteristics of each module:

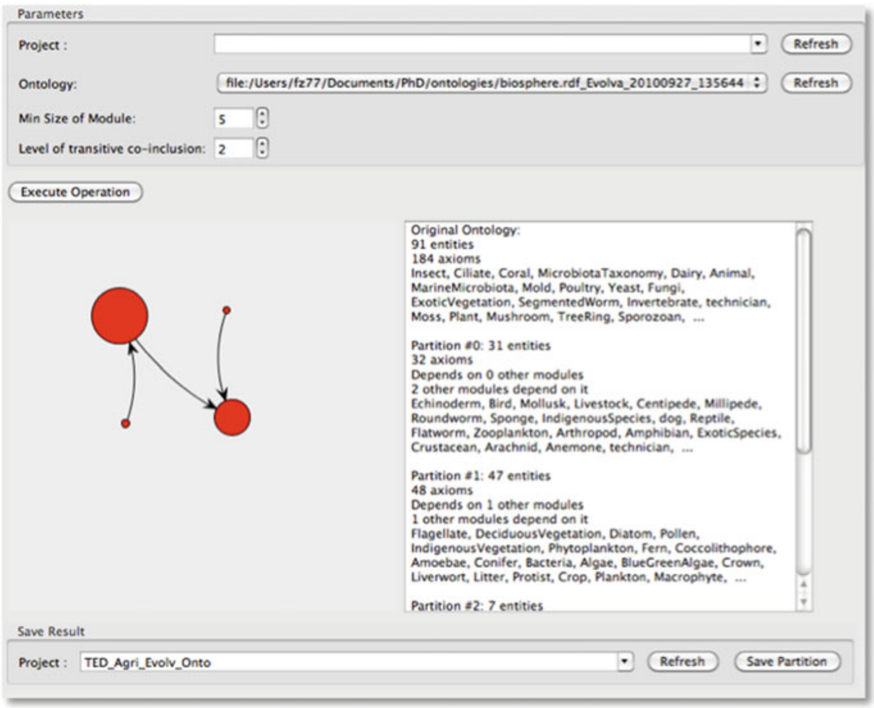
1. *Rule 3 (size of the modules)*: A module should not be smaller than a given threshold. Indeed, initial experiments have shown that applying only the two rules above can result in very small modules. Too small modules can be hard to manage, as it can result in having to consider too many different modules for a given task (e.g. update) (d'Aquin et al. 2007b). Note that, even if it could sometimes be useful, a rule based on the maximum size of a module would not be applicable, as it would contradict rules 1 or 2. In this case, it would be

recommended to use the extraction techniques described in Sect. 10.5.2 to reduce the size of the modules considered too big.

2. *Rule 4 (intra-connectedness)*: Entities within a module should be connected with each other. This is a very simple and natural rule to follow. Indeed, there is no reason for entities that are completely disconnected, directly or indirectly, to end up in the same module.

Having the above rules defined, our algorithm for partitioning ontologies is reasonably straightforward. It basically consists in starting from an initial modularization with as many modules as entities in the ontology. From this initial modularization, the algorithm iteratively enforces rules 1 and 2, merging modules when necessary. At the end of this step, a modularization that respects rules 1, 2 and 4 is obtained. The last task consists in merging modules that are too small according to the given threshold, ensuring that this merging ends up in modules that respect both rules 3 and 4.

Figure 10.8 shows a screenshot of the ontology partitioning plugin integrated with the NeOn Toolkit, which relies on the technique described above. Concretely, this plugin takes the form of a view which allows the user to select the ontology to modularize, specify the threshold for the minimum size of the modules, and execute the algorithm. The result of the algorithm is then presented as a graph, with each



Project :
Ontology: file:/Users/fz77/Documents/PhD/ontologies/biosphere.rdf_Evolva_20100927_135644
Min Size of Module: 5
Level of transitive co-inclusion: 2

Original Ontology:
91 entities
184 axioms
Insect, Ciliate, Coral, MicrobiotaTaxonomy, Dairy, Animal, MarineMicrobiota, Mold, Poultry, Yeast, Fungi, ExoticVegetation, SegmentedWorm, Invertebrate, technician, Moss, Plant, Mushroom, TreeRing, Sporozoan, ...

Partition #0: 31 entities
32 axioms
Depends on 0 other modules
2 other modules depend on it
Echinoderm, Bird, Mollusk, Livestock, Centipede, Millipede, Roundworm, Sponge, IndigenousSpecies, dog, Reptile, Flatworm, Zooplankton, Arthropod, Amphibian, ExoticSpecies, Crustacean, Arachnid, Anemone, technician, ...

Partition #1: 47 entities
48 axioms
Depends on 1 other modules
1 other modules depend on it
Flagellate, DeciduousVegetation, Diatom, Pollen, IndigenousVegetation, Phytoplankton, Fern, Coccolithophore, Amoebae, Conifer, Bacteria, Algae, BlueGreenAlgae, Crown, Liverwort, Litter, Protist, Crop, Plankton, Macrophyte, ...

Partition #2: 7 entities

Project : TED_Agri_Evolv_Onto

Fig. 10.8 Screenshot of the ontology partitioning plugin of the NeOn Toolkit

node corresponding to a created module (details of the module are shown when selecting the corresponding node). The plugin allows the user to save and integrate to the current ontology project each module individually.

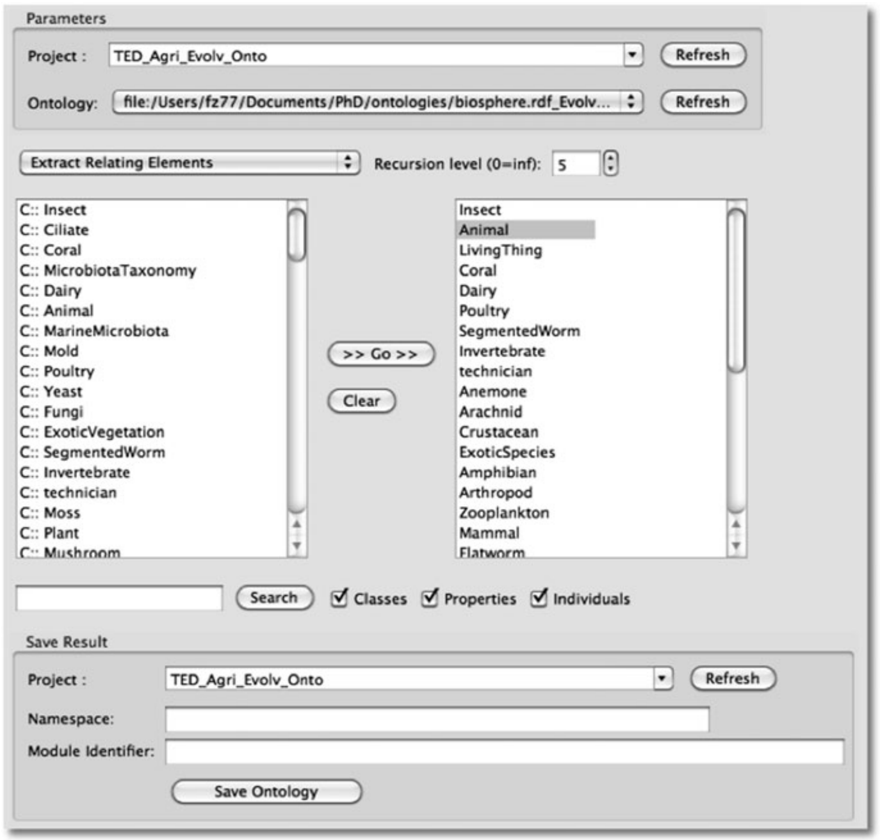
An interesting aspect of the implementation within the NeOn Toolkit is that it allows a very flexible and customizable modularization process. Indeed, it is possible to re-run the algorithm with different parameters, save only the modules that are relevant according to the ontology engineer, and use the module composition plugin presented below to manipulate and customize the modularization until a satisfactory, well-suited modularization is obtained.

10.5.2 Ontology Module Extraction

In d'Aquin et al. (2007a, and 2007b), we have shown through a number of experiments that extracting a module from an ontology is an ill-defined task: the criteria used to decide what should go in a module and what is a good, relevant module are highly dependent on the specificity of the application scenario. In other terms, there is no universal, generic module extraction approach. This appeared also very clearly in the different use cases described in d'Aquin et al. (2008), where different users, in different contexts, provided completely different perspectives about what should go in a module. In general, what appeared from these use cases is that:

1. Users have different, more or less well-defined ideas about what module extraction should do, varying from very elementary cases (e.g. extract a branch) to complex, abstract requirements (should extract everything that helps in interpreting a particular entity). Hence, each of the scenarios we encountered would require a different approach for module extraction.
2. Users want to keep in control of the way the module is created. It is required to support the parameterization of the module extraction for the user to be able to really 'choose' what goes into the module.

For these reasons, we implemented a plugin for the NeOn Toolkit to realize module extraction, providing an interactive and iterative approach to this activity. This plugin integrates a number of different 'operators' for module extraction, most of them being relatively elementary: based on an initial set of entities, extract the super-/sub-classes, entities they depend or that depend on them, common super-/sub-classes, sub-/super-properties, all classes of instances, or all instances of classes. The interface for this plugin (Fig. 10.9) allows the user to easily combine these different elementary operators in an interactive way. An initial module can be created, using particular parameters (here the recursion level), obtaining an initial set of entities to be included. Then another operator can be used, on other entities and other parameters, to refine the module and extend it with other entities until an appropriate module is created. At any point of the process, previous operations can be undone and the module cleared.



Project : TED_Agri_Evolv_Onto

Ontology: file:/Users/fz77/Documents/PhD/ontologies/biosphere.rdf_Evolv...

Extract Relating Elements

Recursion level (0=inf): 5

C:: Insect
C:: Ciliate
C:: Coral
C:: MicrobiotaTaxonomy
C:: Dairy
C:: Animal
C:: MarineMicrobiota
C:: Mold
C:: Poultry
C:: Yeast
C:: Fungi
C:: ExoticVegetation
C:: SegmentedWorm
C:: Invertebrate
C:: technician
C:: Moss
C:: Plant
C:: Mushroom

Insect
Animal
LivingThing
Coral
Dairy
Poultry
SegmentedWorm
Invertebrate
technician
Anemone
Arachnid
Crustacean
ExoticSpecies
Amphibian
Arthropod
Zooplankton
Mammal
Flatworm

[✓ Classes] [✓ Properties] [✓ Individuals]

Project : TED_Agri_Evolv_Onto

Namespace:

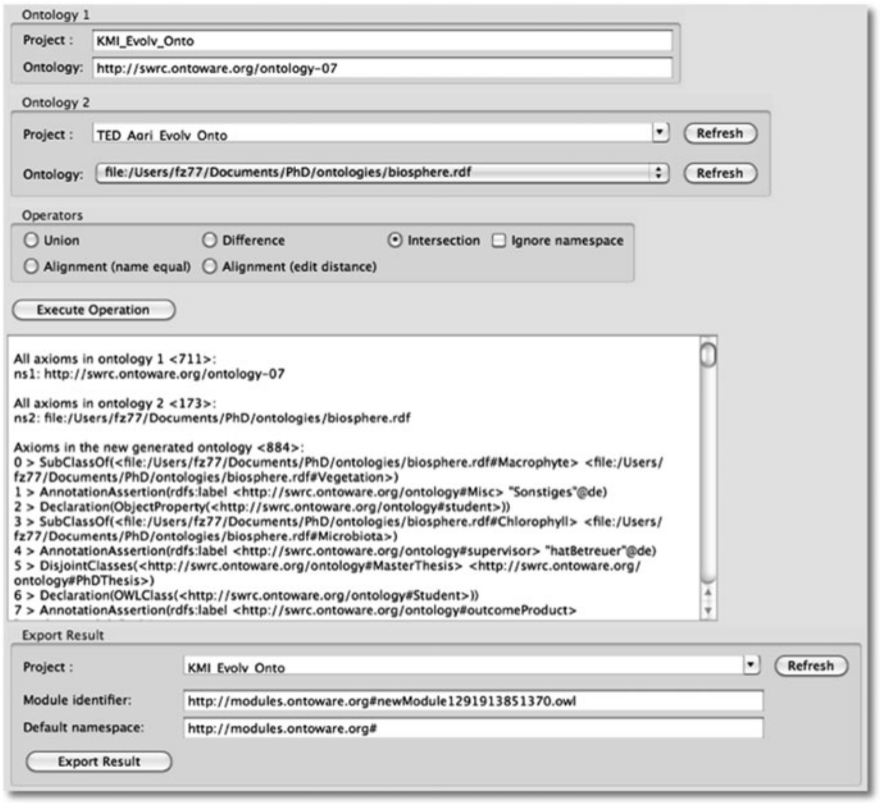
Module Identifier:

Fig. 10.9 Screenshot of the ontology module extraction plugin of the NeOn Toolkit

In addition, the plugin provides straightforward functions to facilitate the selection of the entities to consider for module extraction. This includes restricting the visualization to classes, properties or individuals and searching for entities matching a specific string. Once a module is created, it can simply be saved as part of the current ontology project and become itself processable as an ontology (module) to be composed or partitioned using the other modularization plugins.

10.5.3 Ontology Module Composition

A simple module algebra (including operators for Intersection, Union and Difference of module) is implemented in a dedicated plugin, which is realized as a new NeOn Toolkit view. As shown in Fig. 10.10, in this view, the user selects the two ontologies that serve as input for the operators. In the field between the two


Ontology 1
Project : KMI_Evolv_Onto
Ontology: http://swrc.ontoware.org/ontology-07

Ontology 2
Project : TED Agri Evolv Onto
Ontology: file:/Users/fz77/Documents/PhD/ontologies/biosphere.rdf

Operators
Union · Difference · [✓ Intersection] · Ignore namespace · Alignment (name equal) · Alignment (edit distance)

```
All axioms in ontology 1 <711>:
ns1: http://swrc.ontoware.org/ontology-07

All axioms in ontology 2 <173>:
ns2: file:/Users/fz77/Documents/PhD/ontologies/biosphere.rdf

Axioms in the new generated ontology <884>:
0 > SubClassOf(<file:/Users/fz77/Documents/PhD/ontologies/biosphere.rdf#Macrophyte> <file:/Users/
fz77/Documents/PhD/ontologies/biosphere.rdf#Vegetation>)
1 > AnnotationAssertion(rdfs:label <http://swrc.ontoware.org/ontology#Misc> "Sonstiges"@de)
2 > Declaration(ObjectProperty(<http://swrc.ontoware.org/ontology#student>))
3 > SubClassOf(<file:/Users/fz77/Documents/PhD/ontologies/biosphere.rdf#Chlorophyll> <file:/Users/
fz77/Documents/PhD/ontologies/biosphere.rdf#Microbiota>)
4 > AnnotationAssertion(rdfs:label <http://swrc.ontoware.org/ontology#supervisor> "hatBetreuer"@de)
5 > DisjointClasses(<http://swrc.ontoware.org/ontology#MasterThesis> <http://swrc.ontoware.org/
ontology#PhDThesis>)
6 > Declaration(OWLClass(<http://swrc.ontoware.org/ontology#Student>))
7 > AnnotationAssertion(rdfs:label <http://swrc.ontoware.org/ontology#outcomeProduct>
```

Export Result
Project : KMI Evolv Onto
Module identifier: http://modules.ontoware.org#newModule1291913851370.owl
Default namespace: http://modules.ontoware.org#

Fig. 10.10 Screenshot of the ontology module composition plugin of the NeOn Toolkit

ontologies, the user selects the operator to be applied. In addition to the combination operators, the plugin also supports alignment as an operator, which allows relating modules via mappings. Depending on the operator chosen, the result will be either a new module (for Union, Difference, Intersection) or an alignment (for align).

Finally, the user can specify whether the application of the operators should be sensitive to differences in the namespace. If not, the operators only consider local names. This is for example relevant for the Difference operator applied to two versions of the same ontology – as often, the namespace changes from one version to another (and thus all elements in the ontology), a difference based on the fully qualified names would not be very meaningful.

10.6 Conclusion

In this chapter, we motivated and gave an overview of the activity of ontology modularization. We described a general approach for modularizing ontologies and the tools that have been developed for the NeOn Toolkit ontology engineering environment to support this approach. However, even with the provided tool and methodological support, modularizing an ontology is still a very time-consuming task, not only because of the expensive computation it requires but also because of the expertise and experience needed from the ontology engineer to obtain the desired result (which is very often very hard to establish). We described a simple 'abstract' example of ontology modularization. Further to this work, the empirical analysis of existing modular ontologies and of the process of modularizing existing ontologies could give us further insight into the broad notion of ontology modularity.

References

Cuenca Grau B, Parsia B, Sirin E, Kalyanpur A (2005) Automatic partitioning of owl ontologies using E-connections. In: Description logics, DL2005, Edinburgh

d'Aquin M, Sabou M, Motta E (2006) Modularization: a key for the dynamic selection of relevant knowledge components. In: Workshop on modular ontologies, WoMO 2006, Athens

d'Aquin M, Doran P, Motta E, Tamma V (2007a) Towards a parametric ontology modularization framework based on graph transformation. In: International workshop on modular ontologies, K-CAP 2007, Whistler

d'Aquin M, Schlicht A, Stuckenschmidt H, Sabou M (2007b) Ontology modularization for knowledge selection: experiments and evaluations. In: Database and expert systems applications, 18th international conference, DEXA 2007. Springer, Berlin/Heidelberg/New York

d'Aquin M, Haase P, Rudolph S, Euzenat J, Zimmermann A, Dzbor M, Iglesias M, Jacques Y, Caracciolo C, Buil Aranda C, Gomez, JM (2008) D1.1.3 NeOn formalisms for modularization: syntax, semantics, algebra. NeOn deliverable 1.1.3. NeOn project

d'Aquin M, Schlicht A, Stuckenschmidt H, Sabou M (2009) Criteria and evaluation for ontology modularization technique criteria and evaluation for ontology modularization technique. In: Stuckenschmidt H, Parent C, Spaccapietra S (eds) Modular ontologies: concepts, theories and techniques for knowledge modularization. Springer, Berlin/Heidelberg/New York

Doran P, Tamma V, Iannone L (2007) Ontology module extraction for ontology reuse: an ontology engineering perspective. In: Proceedings of the 2007 ACM CIKM international conference on information and knowledge management, Lisbon

Doran P, Palmisano I, Tamma V (2008) SOMET: algorithm and tool for SPARQL based ontology module extraction. In: International workshop on ontologies: reasoning and modularity (WORM-08), ESWC 2008, Tenerife

Kaushik S, Farkas C, Wijesekera D, Ammann P (2006) An algebra for composing ontologies. In: Formal ontology in information systems, FOIS 2006, Baltimore

Lopez V, Motta E, Dzbor M, d'Aquin M, Peroni S, Guidi D (2009) Final version of the question answering system. Deliverable 8.6 of the OpenKnowledge project

MacCartney B, McIlraith S, Amir E, Uribe TE (2003) Practical partition-based theorem proving for large knowledge bases. In: Proceedings of the international joint conference on artificial intelligence, IJCAI 2003, Acapulco

Melnik S, Rahm E, Bernstein PA (2003) Rondo: a programming platform for generic model management. In: Proceedings of the SIGMOD 2003, San Diego, pp 193–204

Melnik S, Bernstein PA, Halevy AY, Rahm E (2004) A semantics for model management operators. Microsoft technical report

Noy NF, Musen MA (2004) Specifying ontology views by traversal. In: Proceedings of the international semantic web conference, ISWC 2004, Hiroshima

Parnas DL (1978) Designing software for ease of extension and contraction. In: Proceedings of the 3rd international conference on software engineering

Seidenberg J, Rector A (2006) Web ontology segmentation: analysis, classification and use. In: Proceedings of the world wide web conference, WWW 2006, Edinburgh

Stuckenschmidt H (2006) Toward multi-viewpoint reasoning with OWL ontologies. In: Proceedings of the European semantic web conference, ESWC 2006, Budva

Stuckenschmidt H, Klein M (2004) Structure-based partitioning of large concept hierarchies. In: International semantic web conference, ISWC 2004, Hiroshima

Suárez-Figueroa MC (2010) NeOn Methodology for building ontology networks: specification, scheduling and reuse. PhD thesis, Universidad Politécnica de Madrid, España. Available at http://oa.upm.es/3879/

Suntisrivaraporn B, Guilin Q, Ji Q, Haase P (2008) A modularization-based approach to finding all justifications for OWL DL entailments. In: Asian semantic web conference, ASWC 2008, Bangkok

Wiederhold G (1994) An algebra for ontology composition. In: Monterey workshop on formal methods, Monterey